System design: URL Shortener

Suresh Podeti

6 min read · Oct 4, 2023

Introduction

URL shortening is used to create shorter aliases for long URLs. We call these shortened aliases "**short links**". Users are redirected to the original URL when they hit these short links.

Short links **save a lot of space** when displayed, printed, messaged, or tweeted. Additionally, users are less likely to mistype shorter URLs.

For example, if we shorten this page through TinyURL:

https://kittychef.in/restaurants/4a280c7f-1cee-4f51-a52a-8dde3696fa88/menu

We would get:
https://tinyurl.com/2tzhapf5

The shortened URL is nearly one-third the size of the actual URL.
URL shortening is used for **optimizing links across devices**, **tracking individual links** to analyze audience and campaign performance, and **hiding affiliated original URLs.**

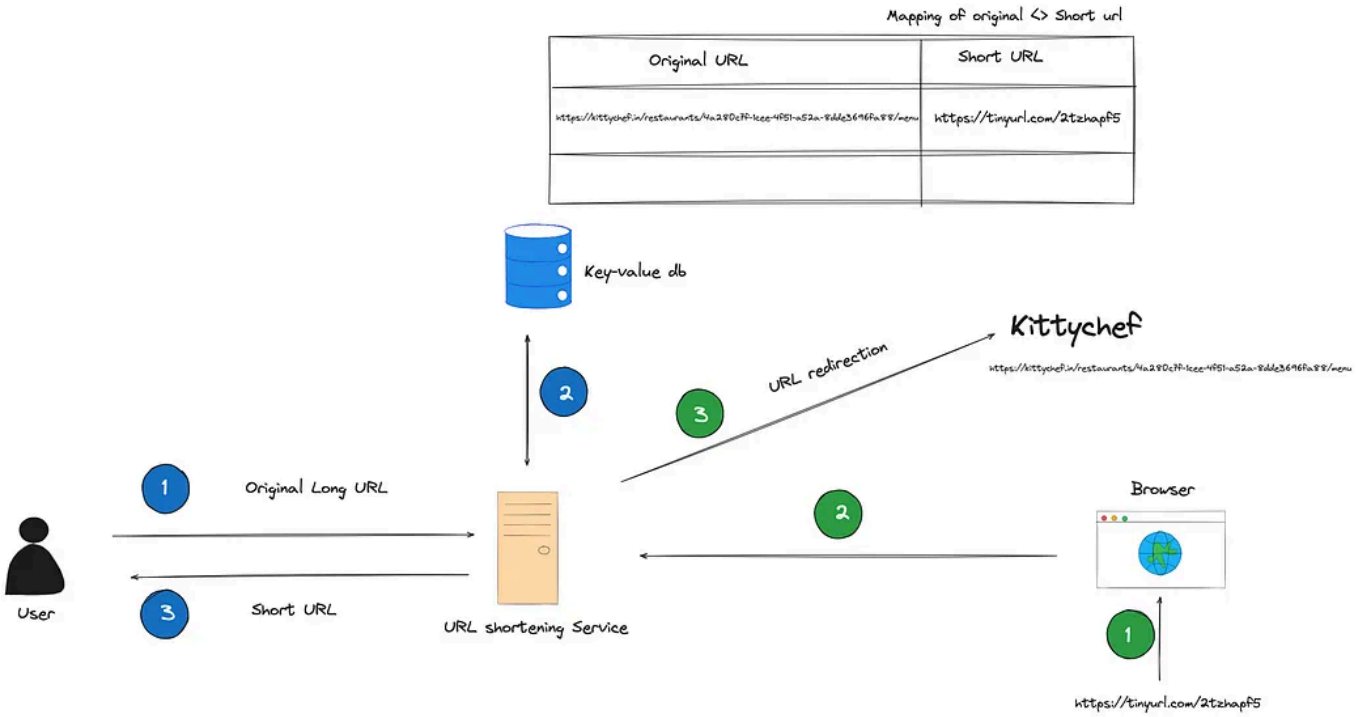

Fig 1.0: How URL shortening service works

Requirements

Functional Requirements

- **Short URL generation:** Our service should be able to generate a unique shorter alias of the given URL.
- **Redirection:** Given a short link, our system should be able to redirect the user to the original URL.

Non-functional Requirements

- **Availability:** Our system should be highly available, because even a fraction of the second downtime w**ould result in URL redirection failures.**
- **Scalability:** Our system should be horizontally scalable with increasing demand.
- **Readability:** The short links generated by our system should be easily readable, distinguishable, and typeable.
- **Latency:** The system should perform at low latency to provide the user with a smooth redirection experience.

Design

Components

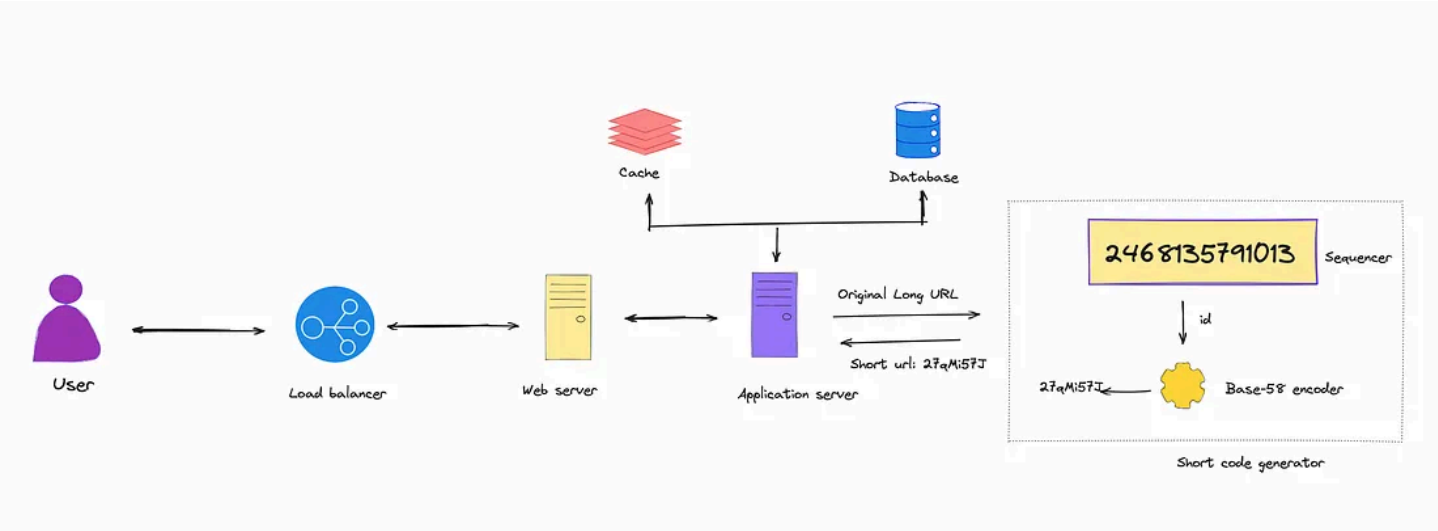

Fig 2.0: A design diagram of the URL shortening service

- **Sequencer** — A sequencer to generate unique IDs.
- **Base-58 encoder** — A Base-58 encoder to enhance the readability of the short URL
- **Database** — We need to store mapping of original long url to short url, and storage has to horizontally scalable. The stored records will have no relationships among themselves, so we don't need structured storage for record-keeping. Considering the reasons above and the fact that our system will be **read-heavy**, NoSQL is a suitable choice for storing data. In particular, **MongoDB** is a good choice for the following reasons — It uses leader-follower protocol, making it possible to use replicas for heavy reading, and it ensures atomicity in concurrent write operations.
- **Cache** — For our specific read-intensive design problem, Memcached is the best choice for a cache solution. We require a simple, horizontally scalable cache system with minimal data structure requirements.

Detailed design

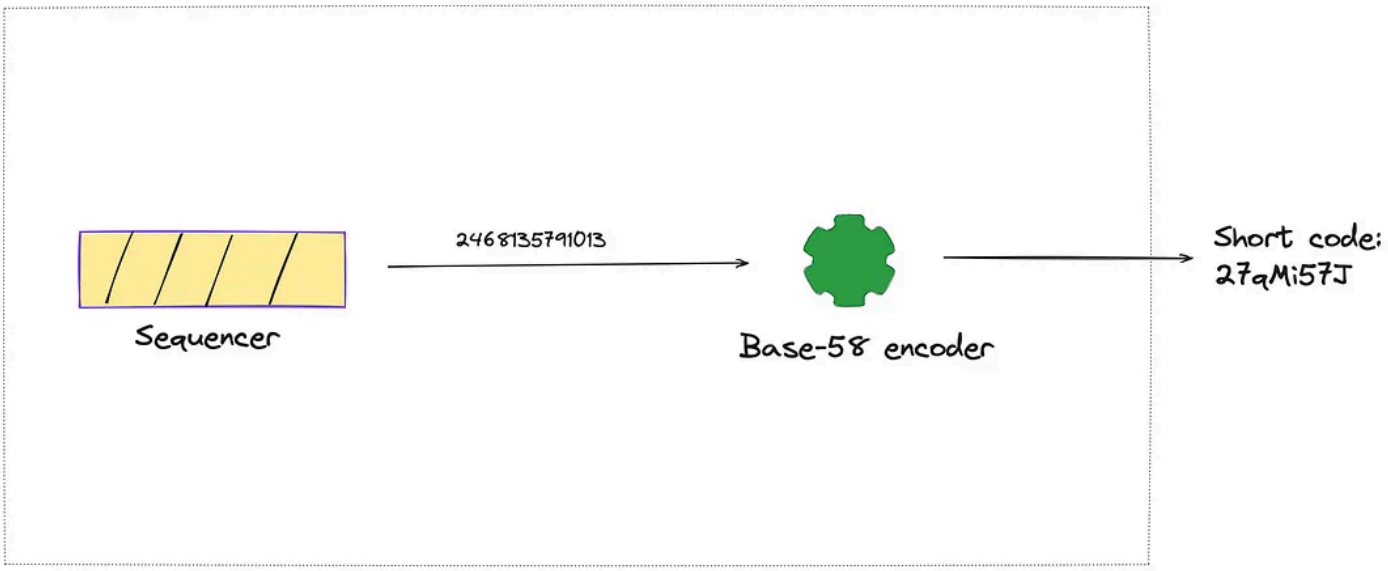

Fig 3.0: Sequencer-encoder workflow

Sequencer

Introduction

medium.com

Encoding (Base 10 → Base 58)

Our sequencer generates a 64-bit ID in base-10, which can be converted to a base-64 short URL. Base-64 is the most common encoding for alphanumeric strings' generation. However, there are some inherent issues with sticking to the base-64 for this design problem: the generated short URL might have **readability issues** because of **look-alike characters.** Characters like `O` (capital o) and `0` (zero), `I` (capital I), and `l` (lower case L) can be confused while characters like `+` and `/` should be avoided because of other system-dependent encodings.

So, we slash out the six characters and use **base-58** instead of base-64 (includes A-Z, a-z, 1–9, `+` and `/`) for enhanced readability purposes. Let's look at our base-58 definition.

{[0,9][A-Z][a-z]+/} — {O, 0, I, l, +, /}

Example: Let's assume that the selected unique ID is `2468135791013` . The following steps show us the remainder calculations:

```
Base-10 = 2468135791013

2468135791013 % 58=17
42554065362 % 58=6
733690782 % 58=4
12649841 % 58=41
218100 % 58=20
3760 % 58=48
64 % 58=6
1 % 58=1
```

Now, we need to write the remainders in order of the most recent to the oldest order.

Base-58 = [1] [6] [48] [20] [41] [4] [6] [17]

Using the table above, we can write the associated characters with the remainders written above.

Base-58 = **27qMi57J**

Sequencer range

Starting range: The minimum default length of the generated short URL should be six characters. Our sequencer can generate a 64-bit binary number that ranges from $1 \rightarrow (2^{64}-1)$. **To meet the requirement for the minimum length of a short URL**, we can select the sequencer IDs to start from at least 10 digits, i.e., **1 Billion.**

Ending point: The maximum number of digits in sequencer IDs that map into the short URL generator's output depends on the maximum utilization of 64 bits, that is, the largest base-10 number in 64-bits. We can estimate the total number of digits in any base by calculating these two points:

1. The numbers of bits to represent one digit in a base-n. This is given by $log2(n)$.
2. Number of digits =*Total bits available / Number of bits to represent one digit*

Let's see the calculations above for both the base-10 and base-58 mathematically:

Base-10: The number of bits needed to represent one decimal digit = $log2(10)$ = 3.133.13

The total number of decimal digits in 64-bit numeric ID = 64 / 3.13 = 20

Base-58:

The number of bits needed to represent one decimal digit = $log2(58)$ = 5.85

The total number of base-58 digits in a 64-bit numeric ID = 64/5.85= 11

The calculations above show that the maximum digits in the sequencer generated ID will be 20 and consequently, the maximum number of characters in the encoded short URL will be **11.**

Sequencer's lifetime

The number of years that our sequencer can provide us with unique IDs depends on two factors:

- Total numbers available in the sequencer = 2^64−10^9 (starting from 1 Billion as discussed above)
- Number of requests per year = 200 Million per month (Assumption) ×12=2.4 Billion

So, taking the above two factors into consideration, we can calculate the expected life of our sequencer.

The lifetime of the sequencer = (2^64−10^9) / 2.4 Billion =**7,686,143,363.63** ***years***

Therefore, our service can run for a long time before the range depletes.

Evaluation

Availability

We need high availability for users generating new short URLs and redirecting them based on the existing short URLs. To handle disasters, we can perform frequent backups of the storage and application servers, and replicate our data sources.

Scalability

Our design is scalable because our data can easily be distributed among horizontally sharded databases. We can employ a consistent hashing scheme to balance the load

between the application and database layers.

Moreover, the large number of unique IDs available in the sequencer's design also ensures the scalability of our system.

Readability

The use of a base-58 encoder, instead of the base-64 encoder, enhances the readability of our system. We divide the readability into two sections:

- **Distinguishable characters** like `0` (zero), `O` (capital o), `I` (capital i), and `l` (lower case L) are eliminated, excluding the possibility of mistaking a character for another look-alike character.
- **Non-alphanumeric characters** like `+` (plus) and `/` (slash) are also eliminated to only have alphanumeric characters in short URLs. Second, it also helps avoid other system-dependent encodings and makes the URLs readily available for all modern file systems and URLs schemes. Such characters may lead to undesired behavior and output during parsing.

This non-functional requirement enhances the user interactivity of our system and makes short URL usage less error-prone for the users.

Latency

Our system ensures low latency with its following features:

- Even the most time-consuming step across the short URL generation process, encoding, takes a few milliseconds. The overall time to generate a short URL is relatively low, ensuring there are no significant delays in this process.
- Our system is redirection-heavy. Writing on the database is minimal compared to reading, and its performance depends on how well it copes with all the redirection requests, compared to the shortening requests. We deliberately chose MongoDB because of its low latency and high throughput in reading-intensive tasks.
- Moreover, the probability of the user using the freshly generated short URL in the next few seconds is relatively low. During this time, synchronous replication to other locations is feasible and therefore adds to the overall low latency of the system for the user.

- The deployment of a distributed cache in our design also ensures that the system redirects the user with the minimum delay possible.

As a result of such design modifications, the system enjoys low latency and high throughput, providing good performance.

Url Shortner System | Software Architect | Software Architecture | Distributed System Design | System Design Interview

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium